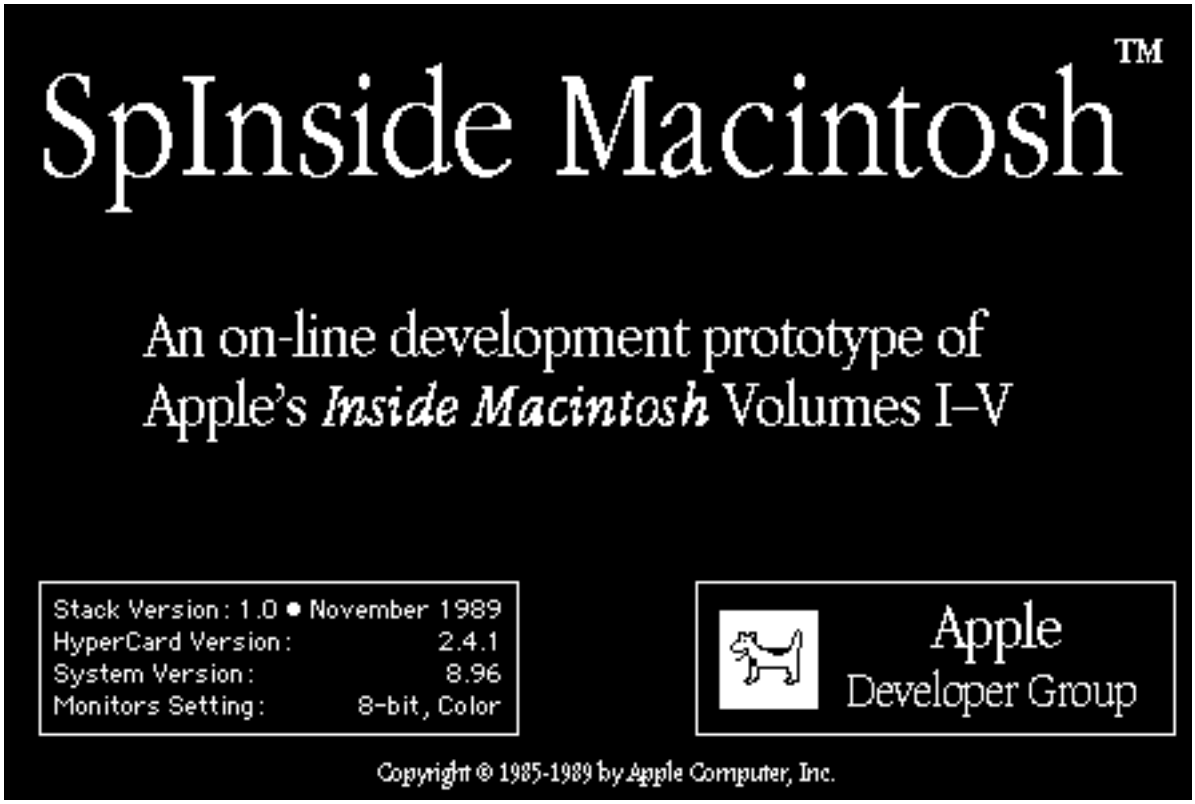

THE SEGMENT LOADER

About This Chapter
About the Segment Loader
Finder Information
Using the Segment Loader
Segment Loader Routines
    Advanced Routines
The Jump Table
Summary of the Segment Loader

ABOUT THIS CHAPTER

This chapter describes the Segment Loader, the part of the Macintosh Operating System that lets you divide your application into several parts and have only some of them in memory at a time. The Segment Loader also provides routines for accessing information about documents that the user has selected to be opened or printed.

You should already be familiar with:

- the basic concepts behind the Resource Manager
- the Memory Manager

ABOUT THE SEGMENT LOADER

The Segment Loader allows you to divide the code of your application into several parts or segments. The Finder starts up an application by calling a Segment Loader routine that loads in the main segment (the one containing the main program). Other segments are loaded in automatically when they're needed. Your application can call the Segment Loader to have these segments removed from memory when they're no longer needed.

The Segment Loader enables you to have programs larger than 32K bytes, the maximum size of a single segment. Also, any code that isn't executed often (such as code for printing) needn't occupy memory when it isn't being used, but can instead be in a separate segment that's "swapped in" when needed.

This mechanism may remind you of the resources of an application, which the Resource Manager reads into memory when necessary. An application's segments are in fact themselves stored as resources; their resource type is 'CODE'. A "loaded" segment has been read into memory by the Resource Manager and locked (so that it's neither relocatable nor purgeable). When a segment is unloaded, it's made relocatable and purgeable.

Every segment has a name. If you do nothing about dividing your program into segments, it will consist only of the main segment. Dividing your program into segments means specifying in your source file the beginning of each segment by name. The names are for your use only; they're not kept around after linking.

FINDER INFORMATION

When the Finder starts up your application, it passes along a list of documents selected by the user to be printed or opened, if any. This information is called the Finder information; its structure is shown in Figure 1.

It's up to your application to access the Finder information and open or print the documents selected by the user.

The message in the first word of the Finder information indicates whether the documents are to be opened (0) or printed (1), and the count following it indicates the number of documents (0 if none). The rest of the Finder information specifies each of the selected documents by volume reference number, file type, version number, and file name; these terms are explained in the File Manager chapter and the Finder Interface chapter. File names are padded to an even number of bytes if necessary.

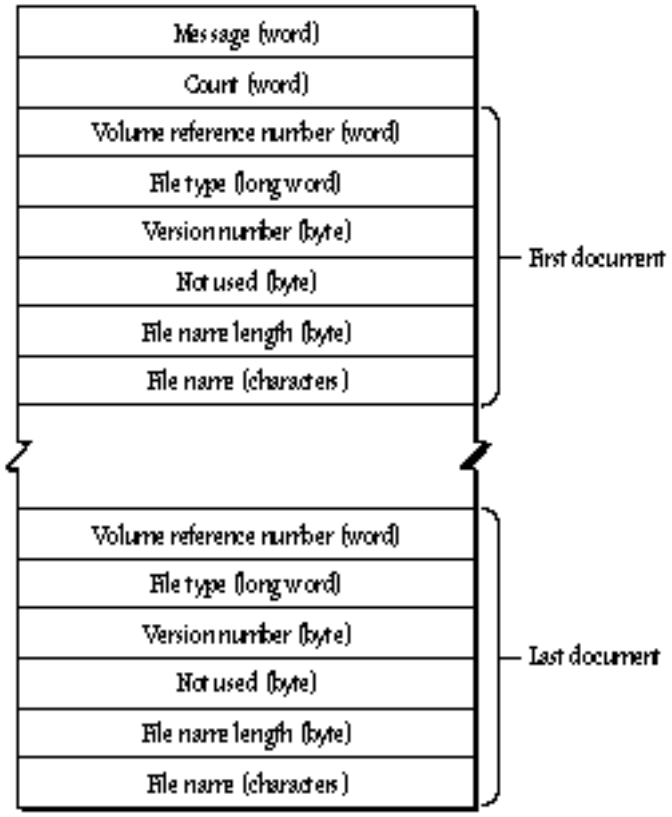

Figure 1–Finder Information

Figure 1–Finder Information

Your application should start up with an empty untitled document on the desktop if there are no documents listed in the Finder information. If one or more documents are to be opened, your application should go through each document one at a time, and determine whether it can be opened. If it can be opened, you should do so, and then check the next document in the list (unless you've opened your maximum number of documents, in which case you should ignore the rest). If your application doesn't recognize a document's file type (which can happen if the user selected your application along with another application's document), you may want to open the document anyway and check its internal structure to see if it's a compatible type. Display an alert box including the name of each document that can't be opened.

If one or more documents are to be printed, your application should go through each document in the list and determine whether it can be printed. If any documents can be printed, the application should display the standard Print dialog box and then print each document—preferably without doing its entire startup sequence. For example, it may not be necessary to show the menu bar or the document window. If the document can't be printed, ignore it; it may be intended for another application.

USING THE SEGMENT LOADER

When your application starts up, you should determine whether any documents were selected to be printed or opened by it. First call CountAppFiles, which returns the number of selected documents and indicates whether they're to be printed or opened. If the number of selected documents is 0, open an empty untitled document in the normal manner. Otherwise, call GetAppFiles once for each selected document. GetAppFiles returns information about each document, including its file type. Based on the file type, your application can decide how to treat the document, as described in the preceding section. For each document that your application opens or prints, call ClrAppFiles, which indicates to the Finder that you've processed it.

To unload a segment when it's no longer needed, call UnloadSeg. If you don't want to keep track of when each particular segment should be unloaded, you can call UnloadSeg for every segment in your application at the end of your main event loop. This isn't harmful, since the segments aren't purged unless necessary.

Note: The main segment is always loaded and locked.

Warning: A segment should never unload the segment that called it, because the return addresses on the stack would refer to code that may be moved or purged.

Another procedure, GetAppParms, lets you get information about your application such as its name and the reference number for its resource file. The Segment Loader also provides the ExitToShell procedure—a way for an application to quit and return the user to the Finder.

Finally, there are three advanced routines that can be called only from assembly language: Chain, Launch, and LoadSeg. Chain starts up another application without disturbing the application heap. Thus the current application can let another application take over while still keeping its data around in the heap. Launch is called by the Finder to start up an application; it's like Chain but doesn't retain the application heap. LoadSeg is called indirectly (via the jump table, as described later) to load segments when necessary—that is, whenever a routine in an unloaded segment is invoked.

Assembly-language note: Instead of using CountAppFiles, GetAppFiles, and ClrAppFiles, assembly-language programmers can access the Finder information via the global variable AppParmHandle, which contains a handle to the Finder information. Parse the Finder information as shown in Figure 1 above. For each document that your application opens or prints, set the file type in the Finder information to 0.

```
PROCEDURE CountAppFiles (VAR message:  INTEGER;
                         VAR count:  INTEGER); [Not in ROM]
```

CountAppFiles deciphers the Finder information passed to your application, and returns information about the documents that were selected when your application started up. It returns the number of selected documents in the count parameter, and a number in the message parameter that indicates whether the documents are to opened or printed:

```
CONST  appOpen  = 0;  {open the document(s)}
       appPrint = 1;  {print the document(s)}
```

```
PROCEDURE GetAppFiles (index:  INTEGER; VAR theFile:  AppFile); [Not in ROM]
```

GetAppFiles returns information about a document that was selected when your application started up (as listed in the Finder information). The index parameter indicates the file for which information should be returned; it must be between 1 and the number returned by CountAppFiles, inclusive. The information is returned in the following data structure:

```
TYPE  AppFile = RECORD
                  vRefNum:  INTEGER;    {volume reference number}
                  fType:    OSType;     {file type}
                  versNum:  INTEGER;    {version number}
                  fName:    Str255      {file name}
                END;
```

```
PROCEDURE ClrAppFiles (index:  INTEGER); [Not in ROM]
```

ClrAppFiles changes the Finder information passed to your application about the specified file such that the Finder knows you've processed the file. The index parameter must be between 1 and the number returned by CountAppFiles. You should call ClrAppFiles for every document your application opens or prints, so that the information returned by CountAppFiles and GetAppFiles is always correct. (ClrAppFiles sets the file type in the Finder information to 0.)

```
PROCEDURE GetAppParms (VAR apName:  Str255; VAR apRefNum:  INTEGER;
                       VAR apParam:  Handle);
```

GetAppParms returns information about the current application. It returns the application name in apName and the reference number for the application's resource file in apRefNum. A handle to the Finder information is returned in apParam, but the Finder information is more easily accessed with the GetAppFiles call.

Assembly-language note: Assembly-language programmers can instead get the application name, reference number, and handle to the Finder information directly from the global variables CurApName, CurApRefNum, and AppParmHandle.

Note: If you simply want the application's resource file reference number, you can call the Resource Manager function CurResFile when the application starts up.

```
PROCEDURE UnloadSeg (routineAddr:  Ptr);
```

UnloadSeg unloads a segment, making it relocatable and purgeable; routineAddr is the address of any externally referenced routine in the segment. The segment won't actually be purged until the memory it occupies is needed. If the segment is purged, the Segment Loader will reload it the next time one of the routines in it is called.

Note: UnloadSeg will work only if called from outside the segment to be unloaded.

```
PROCEDURE ExitToShell;
```

ExitToShell provides an exit from an application by starting up the Finder (after releasing the entire application heap).

Assembly-language note: ExitToShell actually launches the application whose name is stored in the global variable FinderName.

Advanced Routines

The routines below are provided for advanced programmers; they can be called only from assembly language.

Chain procedure

```
Trap macro  _Chain
On entry     (A0):  pointer to application's file name (preceded by length byte)
            4(A0):  configuration of sound and screen buffers (word)
```

Chain starts up an application without doing anything to the application heap, so the current application can let another application take over while still keeping its data around in the heap.

Chain also configures memory for the sound and screen buffers. The value you pass in 4(A0) determines which sound and screen buffers are allocated:

- If you pass 0 in 4(A0), you get the main sound and screen buffers; in this

case, you have the largest amount of memory available to your application.
- Any positive value in 4(A0) causes the alternate sound buffer and main screen buffer to be allocated.
- Any negative value in 4(A0) causes the alternate sound buffer and alternate screen buffer to be allocated.

The memory map in the Memory Manager chapter shows the locations of the screen and sound buffers.

Warning: The sound buffers and alternate screen buffer are only supported on the Macintosh 128K, 512K (and enhanced), Plus, and SE models.

Note: You can get the most recent value passed in 4(A0) to the Chain procedure from the global variable CurPageOption.

Chain closes the resource file for any previous application and opens the resource file for the application being started; call DetachResource for any resources that you still wish to access.

Launch procedure

•••Click on the X-Ref button, and refer to Technical Note #126.•••

```
Trap macro  _Launch
On entry    (A0):  pointer to application's file name (preceded by length byte)
           4(A0):  configuration of sound and screen buffers (word)
```

Launch is called by the Finder to start up an application and will rarely need to be called by an application itself. It's the same as the Chain procedure (described above) except that it frees the storage occupied by the application heap and restores the heap to its original size.

Note: Launch preserves a special handle in the application heap which is used for preserving the desk scrap between applications; see the Scrap Manager chapter for details.

LoadSeg procedure

```
Trap macro  _LoadSeg
On entry    stack:  segment number (word)
```

LoadSeg is called indirectly via the jump table (as described in the following section) when the application calls a routine in an unloaded segment. It loads the segment having the given segment number, which was assigned by the Linker. If the segment isn't in memory, LoadSeg calls the Resource Manager to read it in. It changes the jump table entries for all the routines in the segment from the "unloaded" to the "loaded" state and then invokes the routine that was called.

Note: Since LoadSeg is called via the jump table, there isn't any need for you to call it yourself.

Advanced programmers: The LoadSeg procedure has been modified to help reduce heap fragmentation. If the code segment to be loaded is unlocked (that is, if it's not in memory and its

resLocked attribute is clear, or if it is in memory and is unlocked), LoadSeg calls the Memory Manager procedure MoveHHi to move the segment toward the top of the current heap zone.

To maintain compatibility with the 64K ROM, your code segments should be locked in the resource file. They will, however, be unlocked when they're unloaded and may float up in the heap; subsequent loading may then cause heap fragmentation.

If your application will never run on a 64K ROM machine, all segments except the main segment ('CODE' resource 1) can be unlocked in the resource file. Your application's initialization routine must call the Memory Manager procedure MaxApplZone, however; otherwise the heap zone will grow incrementally and calls to MoveHHi may leave your segments scattered throughout the heap.

THE JUMP TABLE

This section describes how the Segment Loader works internally, and is included here for advanced programmers; you don't have to know about this to be able to use the common Segment Loader routines.

The loading and unloading of segments is implemented through the application's jump table. The jump table contains one eight-byte entry for every externally referenced routine in every segment; all the entries for a particular segment are stored contiguously. The location of the jump table is shown in the Memory Manager chapter.

When the Linker encounters a call to a routine in another segment, it creates a jump table entry for the routine (see Figure 2). The jump table refers to segments by segment numbers assigned by the Linker. If the segment is loaded, the jump table entry contains code that jumps to the routine. If the segment isn't loaded, the entry contains code that loads the segment.

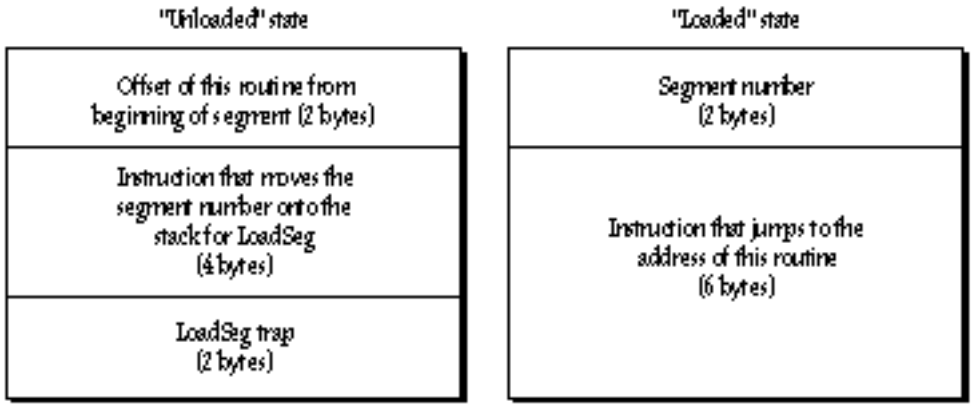

Figure 2–Format of a Jump Table Entry

Figure 2–Format of a Jump Table Entry

•••Click on the X-Ref button, and refer to Technical Note #220.•••

When a segment is unloaded, all its jump table entries are in the "unloaded" state. When a call to a routine in an unloaded segment is made, the code in the last six bytes of its jump table entry is executed. This code calls LoadSeg, which loads the segment into memory, transforms all of its jump table entries to the "loaded" state, and invokes the routine by executing the instruction in the last six bytes of the jump table entry. Subsequent calls to the routine also execute this instruction. If UnloadSeg is called to unload the segment, it restores the jump table entries to their "unloaded" state. Notice that whether the segment is loaded or unloaded, the last six bytes of the jump table entry are executed; the effect depends on the state of the entry at the time.

To be able to set all the jump table entries for a segment to a particular state, LoadSeg and UnloadSeg need to know exactly where in the jump table all the entries are located. They get this information from the segment header, four bytes at the beginning of the segment which contain the following:

Number of bytes	Contents
2 bytes	Offset of the first routine's entry from the beginning of the jump table
2 bytes	Number of entries for this segment

When an application starts up, its jump table is read in from segment 0 (which is the 'CODE' resource with an ID of 0). This is a special segment created by the Linker for every executable file. It contains the following:

Number of bytes	Contents
4 bytes	"Above A5" size; size in bytes from location pointed to by A5 to upper end of application space
4 bytes	"Below A5" size; size in bytes of application globals plus QuickDraw globals
4 bytes	Length of jump table in bytes
4 bytes	Offset to jump table from location pointed to by A5
n bytes	Jump table

Note: For all applications, the offset to the jump table from the location pointed to by A5 is 32, and the "above A5" size is 32 plus the length of the jump table.

The Segment Loader then executes the first entry in the jump table, which loads the main segment ('CODE' resource 1) and starts the application.

Assembly-language note: The offset to the jump table from the location pointed to by A5 is stored in the global variable CurJTOffset.

SUMMARY OF THE SEGMENT LOADER

Constants

```
CONST
  { Message returned by CountAppleFiles }

  appOpen  = 0;  {open the document(s)}
  appPrint = 1;  {print the document(s)}
```

Data Types

```
TYPE
  AppFile = RECORD
              vRefNum:  INTEGER;    {volume reference number}
              fType:    OSType;     {file type}
              versNum:  INTEGER;    {version number}
              fName:    Str255      {file name}
            END;
```

Routines

```
PROCEDURE CountAppFiles  (VAR message:  INTEGER; VAR count:  INTEGER);
                         [Not in ROM]
PROCEDURE GetAppFiles    (index:  INTEGER; VAR theFile:  AppFile); [Not in ROM]
PROCEDURE ClrAppFiles    (index:  INTEGER); [Not in ROM]
PROCEDURE GetAppParms    (VAR apName:  Str255; VAR apRefNum:  INTEGER;
                          VAR apParam:  Handle);
PROCEDURE UnloadSeg      (routineAddr:  Ptr);
PROCEDURE ExitToShell;
```

Assembly-Language Information

Advanced Routines

```
Trap macro  On entry
_Chain       (A0):  pointer to application's file name (preceded by length byte)
            4(A0):  configuration of sound and screen buffers (word)
_Launch      (A0):  pointer to application's file name (preceded by length byte)
            4(A0):  configuration of sound and screen buffers (word)
_LoadSeg     stack: segment number (word)
```

Variables

```
AppParmHandle  Handle to Finder information
CurApName      Name of current application (length byte followed
```

	by up to 31 characters)
CurApRefNum	Reference number of current application's resource file (word)
CurPageOption	Sound/screen buffer configuration passed to Chain or Launch (word)
CurJTOffset	Offset to jump table from location pointed to by A5 (word)
FinderName	Name of the Finder (length byte followed by up to 15 characters)

Further Reference:

Resource Manager
Memory Manager
Technical Note #43, Calling LoadSeg
Technical Note #113, Boot Blocks
Technical Note #126, Sub(Launching) from a High-Level Language
Technical Note #220, Segment Loader Limitations
Technical Note #239, Inside Object Pascal
Technical Note #240, Using MPW for Non-Macintosh 68000 Systems
Technical Note #256, Globals in Stand-Alone Code?

END OF DOCUMENT